The Manticore Approach to Parallelism

John Reppy

University of Chicago

October 14, 2009

Hardware (r)evolution

The hardware environment is heterogeneous and in flux:

- Microprocessors have become multiprocessors.
- Quad-core is standard on the desktop; 8-core by year's end.
- Larrabee is rumored to have 32 cores
- Heterogeneous processors (Cell and GPU)

Writing parallel programs in this environment is more challenging than ever!

And it is a problem for everyone!

Hardware (r)evolution

The hardware environment is heterogeneous and in flux:

- Microprocessors have become multiprocessors.
- Quad-core is standard on the desktop; 8-core by year's end.
- Larrabee is rumored to have 32 cores
- Heterogeneous processors (Cell and GPU)

Writing parallel programs in this environment is more challenging than ever!

And it is a problem for everyone!

Hardware (r)evolution

The hardware environment is heterogeneous and in flux:

- Microprocessors have become multiprocessors.
- Quad-core is standard on the desktop; 8-core by year's end.
- Larrabee is rumored to have 32 cores
- Heterogeneous processors (Cell and GPU)

Writing parallel programs in this environment is more challenging than ever!

And it is a problem for everyone!

Hardware (r)evolution

The hardware environment is heterogeneous and in flux:

- Microprocessors have become multiprocessors.
- Quad-core is standard on the desktop; 8-core by year's end.
- Larrabee is rumored to have 32 cores
- Heterogeneous processors (Cell and GPU)

Writing parallel programs in this environment is more challenging than ever!

And it is a problem for everyone!

Hardware (r)evolution

The hardware environment is heterogeneous and in flux:

- Microprocessors have become multiprocessors.
- Quad-core is standard on the desktop; 8-core by year's end.
- Larrabee is rumored to have 32 cores
- Heterogeneous processors (Cell and GPU)

Writing parallel programs in this environment is more challenging than ever!

And it is a problem for everyone!

Hardware (r)evolution

The hardware environment is heterogeneous and in flux:

- Microprocessors have become multiprocessors.
- Quad-core is standard on the desktop; 8-core by year's end.
- Larrabee is rumored to have 32 cores
- Heterogeneous processors (Cell and GPU)

Writing parallel programs in this environment is more challenging than ever!

And it is a problem for everyone!

The Manticore project

The Manticore project is motivated by the need for parallelism in commodity applications.

- Need high-level constructs to hide hardware details
- Support for heterogeneous applications
- Opportunity for functional programming (again)
- Challenge: efficient implementation on a range of hardware platforms.

NOTE: we have purposefully avoided HPC as an application area!

The Manticore project

The Manticore project is motivated by the need for parallelism in commodity applications.

- Need high-level constructs to hide hardware details
- Support for heterogeneous applications
- Opportunity for functional programming (again)
- Challenge: efficient implementation on a range of hardware platforms.

NOTE: we have purposefully avoided HPC as an application area!

The Manticore project

The Manticore project is motivated by the need for parallelism in commodity applications.

- Need high-level constructs to hide hardware details
- Support for heterogeneous applications
- Opportunity for functional programming (again)
- Challenge: efficient implementation on a range of hardware platforms.

NOTE: we have purposefully avoided HPC as an application area!

The Manticore project

The Manticore project is motivated by the need for parallelism in commodity applications.

- Need high-level constructs to hide hardware details
- Support for heterogeneous applications
- Opportunity for functional programming (again)
- Challenge: efficient implementation on a range of hardware platforms.

NOTE: we have purposefully avoided HPC as an application area!

The Manticore project

The Manticore project is motivated by the need for parallelism in commodity applications.

- Need high-level constructs to hide hardware details
- Support for heterogeneous applications
- Opportunity for functional programming (again)
- Challenge: efficient implementation on a range of hardware platforms.

NOTE: we have purposefully avoided HPC as an application area!

The Manticore project

The Manticore project is motivated by the need for parallelism in commodity applications.

- Need high-level constructs to hide hardware details
- Support for heterogeneous applications
- Opportunity for functional programming (again)
- Challenge: efficient implementation on a range of hardware platforms.

NOTE: we have purposefully avoided HPC as an application area!

People

The Manticore project is a joint project between the University of Chicago and the Rochester Institute of Technology.

Lars Bergstrom	*University of Chicago*
Matthew Fluet	*Rochester Institute of Technology*
Mike Rainey	*University of Chicago*
Adam Shaw	*University of Chicago*
Yingqi Xiao	*University of Chicago*

with help from

Nic Ford, Korei Klein, Joshua Knox, Jon Riehl, Ridge Scott at the University of Chicago

Also, thanks to the NSF for funding this research.

Language design

Our initial design is purposefully conservative. It can be summarized as the combination of three distinct sub-languages:

- A mutation-free subset of SML (no refs or arrays, but includes exceptions).
- Language mechanisms for implicitly-threaded parallel programming.
- Language mechanisms for explicitly-threaded parallel programming (*a.k.a.* concurrent programming) based on message passing (not MPI).

The focus of this talk will be on the implicitly-threaded mechanisms.

Language design

Our initial design is purposefully conservative. It can be summarized as the combination of three distinct sub-languages:

- A mutation-free subset of SML (no refs or arrays, but includes exceptions).
- Language mechanisms for implicitly-threaded parallel programming.
- Language mechanisms for explicitly-threaded parallel programming (*a.k.a.* concurrent programming) based on message passing (not MPI).

The focus of this talk will be on the implicitly-threaded mechanisms.

Language design

Our initial design is purposefully conservative. It can be summarized as the combination of three distinct sub-languages:

- A mutation-free subset of SML (no refs or arrays, but includes exceptions).
- Language mechanisms for implicitly-threaded parallel programming.
- Language mechanisms for explicitly-threaded parallel programming (*a.k.a.* concurrent programming) based on message passing (not MPI).

The focus of this talk will be on the implicitly-threaded mechanisms.

Language design

Our initial design is purposefully conservative. It can be summarized as the combination of three distinct sub-languages:

- A mutation-free subset of SML (no refs or arrays, but includes exceptions).
- Language mechanisms for implicitly-threaded parallel programming.
- Language mechanisms for explicitly-threaded parallel programming (*a.k.a.* concurrent programming) based on message passing (not MPI).

The focus of this talk will be on the implicitly-threaded mechanisms.

Language design

Our initial design is purposefully conservative. It can be summarized as the combination of three distinct sub-languages:

- A mutation-free subset of SML (no refs or arrays, but includes exceptions).
- Language mechanisms for implicitly-threaded parallel programming.
- Language mechanisms for explicitly-threaded parallel programming (*a.k.a.* concurrent programming) based on message passing (not MPI).

The focus of this talk will be on the implicitly-threaded mechanisms.

Language design *(continued ...)*

Manticore provides several light-weight syntactic forms for introducing parallel computation.

These forms are declarative and are treated as hints by the system.

- Parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancelation of unused subcomputations.
- Parallel case provides non-deterministic speculative parallelism.

Language design *(continued ...)*

Manticore provides several light-weight syntactic forms for introducing parallel computation.

These forms are declarative and are treated as hints by the system.

- Parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancelation of unused subcomputations.
- Parallel case provides non-deterministic speculative parallelism.

Language design *(continued ...)*

Manticore provides several light-weight syntactic forms for introducing parallel computation.

These forms are declarative and are treated as hints by the system.

- Parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancelation of unused subcomputations.
- Parallel case provides non-deterministic speculative parallelism.

Language design *(continued ...)*

Manticore provides several light-weight syntactic forms for introducing parallel computation.

These forms are declarative and are treated as hints by the system.

- Parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancelation of unused subcomputations.
- Parallel case provides non-deterministic speculative parallelism.

Language design *(continued ...)*

Manticore provides several light-weight syntactic forms for introducing parallel computation.

These forms are declarative and are treated as hints by the system.

- Parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancelation of unused subcomputations.
- Parallel case provides non-deterministic speculative parallelism.

Parallel arrays

We support fine-grained nested-data-parallel (NDP) computation using a parallel array comprehension form (NESL/Nepal/DPH):

$$\texttt{[|}\ exp\ \texttt{|}\ pat_i\ \texttt{in}\ exp_i\ \texttt{where}\ pred\ \texttt{|]}$$

For example, the parallel point-wise summing of two arrays:

```
[| x+y | x in xs, y in ys |]
```

NOTE: zip semantics, not Cartesian-product semantics.

This construct can be mapped onto SIMD type hardware (GPUs).

Nested data parallelism *(continued ...)*

Mandelbrot set computation:

```
fun x i = x0 + dx * Float.fromInt i;
fun y j = y0 - dy * Float.fromInt j;
fun loop (cnt, re, im) =
      if (cnt < 255) andalso (re*re + im*im > 4.0)
        then loop(cnt+1, re*re - re*im + re, 2.0*re*im + im)
        else cnt;
[|
  [| loop(0, x i, y j) | i in [| 0..N |] |]
  | j in [| 0..N |]
|]
```

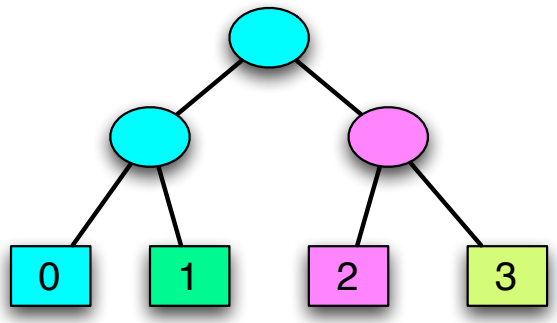
Irregular parallelism

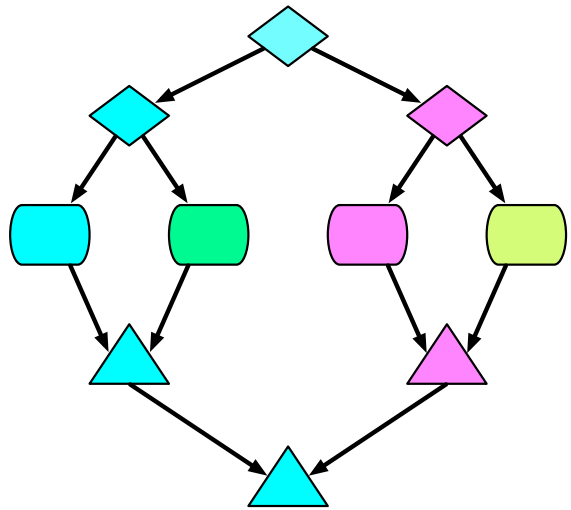
```
type sparse_matrix = (int * float) parray parray

fun sparseDotP (sv, v) = sumP [| x * v!i | (i, x) in sv |]

fun smxv (sm, v) = [| sparseDotP(row, v) | row in sm |]
```

Parallel tuples

Parallel tuples provide fork-join parallelism. For example, consider summing the leaves of a binary tree.

```
datatype tree = LF of long | ND of tree * tree

fun treeAdd (LF n) = n
  | treeAdd (ND(t1, t2)) =
          (op +) (| treeAdd t1, treeAdd t2 |)
```





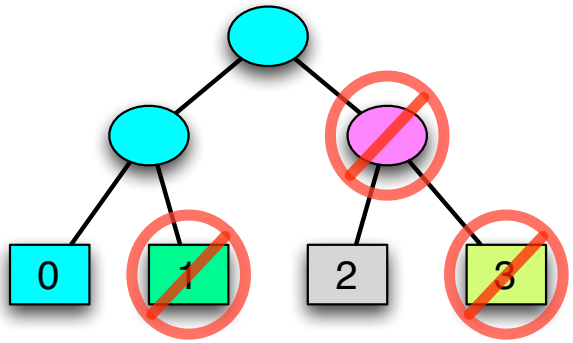
Parallel bindings

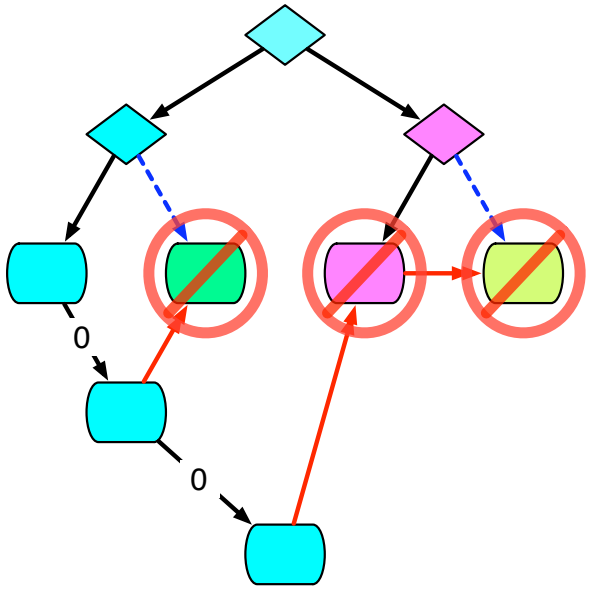
Parallel bindings provide more flexibility than parallel tuples. For example, consider computing the product of the leaves of a binary tree.

```
fun treeMul (LF n) = n
  | treeMul (ND(t1, t2)) = let
      pval b = treeMul t2
      val  a = treeMul t1
      in
        if (a = 0) then 0 else a*b
      end
```

NOTE: the computation of `b` is speculative.

Parallel bindings *(continued ...)*





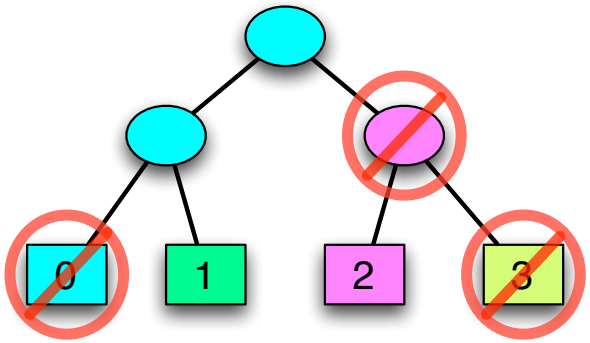
Parallel case

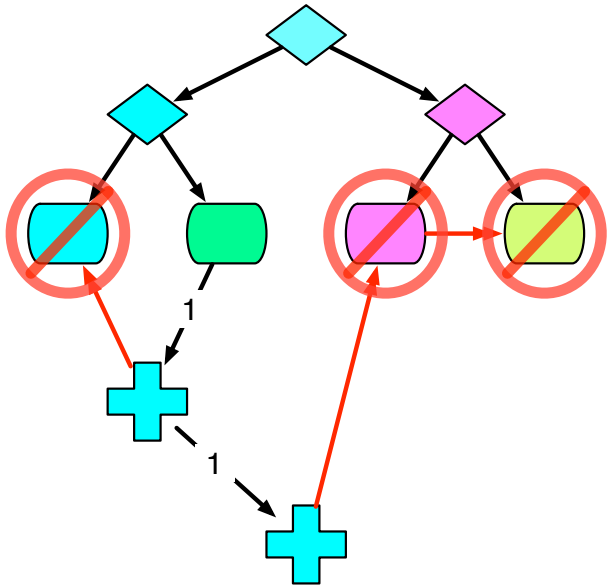
Parallel case supports speculative parallelism when we want the quickest answer (e.g., search problems). For example, consider picking a leaf of the tree:

```
fun treePick (LF n) = n
  | treePick (ND(t1, t2)) = (
      pcase treePick t1 & treePick t2
        of ? & n => n
         | n & ? => n)
```

There is some similarity with join patterns.

Parallel case *(continued ...)*





Parallel case *(continued ...)*

Symmetric version of `treeMul`.

```
fun treeMul (LF n) = n
  | treeMul (ND(t1, t2)) = (
      pcase treeMul t1 & treeMul t2
        of ? & 0 => 0
         | 0 & ? => 0
         | a & b => a*b)
```

Discussion

- These mechanisms compose.
- Value-oriented computation model
- Use tree-structure (ropes) for `parray` type.
- Futures with Cilk-style work stealing plus cancelation
- Working on size analysis to manage granularity

Discussion

- These mechanisms compose.
- Value-oriented computation model
- Use tree-structure (ropes) for `parray` type.
- Futures with Cilk-style work stealing plus cancelation
- Working on size analysis to manage granularity

Discussion

- These mechanisms compose.
- Value-oriented computation model
- Use tree-structure (ropes) for `parray` type.
- Futures with Cilk-style work stealing plus cancelation
- Working on size analysis to manage granularity

Discussion

- These mechanisms compose.
- Value-oriented computation model
- Use tree-structure (ropes) for `parray` type.
- Futures with Cilk-style work stealing plus cancelation
- Working on size analysis to manage granularity

Discussion

- These mechanisms compose.
- Value-oriented computation model
- Use tree-structure (ropes) for `parray` type.
- Futures with Cilk-style work stealing plus cancelation
- Working on size analysis to manage granularity

Status

- The project is about three years old.
- We have a prototype implementation for the x86-64 processor (Linux and Mac OS X).
- Demonstrated scalable performance on 16-core system (4 quad-core AMD 8380 processors) vs. good sequential implementations.
- Sequential performance is okay, but needs improvement.

Status

- The project is about three years old.
- We have a prototype implementation for the x86-64 processor (Linux and Mac OS X).
- Demonstrated scalable performance on 16-core system (4 quad-core AMD 8380 processors) vs. good sequential implementations.
- Sequential performance is okay, but needs improvement.

Status

- The project is about three years old.
- We have a prototype implementation for the x86-64 processor (Linux and Mac OS X).
- Demonstrated scalable performance on 16-core system (4 quad-core AMD 8380 processors) vs. good sequential implementations.
- Sequential performance is okay, but needs improvement.

Status

- The project is about three years old.
- We have a prototype implementation for the x86-64 processor (Linux and Mac OS X).
- Demonstrated scalable performance on 16-core system (4 quad-core AMD 8380 processors) vs. good sequential implementations.
- Sequential performance is okay, but needs improvement.

Future work

- Applications: interactive graphics, computer vision, and medical imaging.
- Mapping NDP constructs onto GPUs.
- Better support for speculative parallelism.
- Support for controlled use of mutation for shared data structures.
- Ongoing work to improve sequential performance.

Future work

- Applications: interactive graphics, computer vision, and medical imaging.
- Mapping NDP constructs onto GPUs.
- Better support for speculative parallelism.
- Support for controlled use of mutation for shared data structures.
- Ongoing work to improve sequential performance.

Future work

- Applications: interactive graphics, computer vision, and medical imaging.
- Mapping NDP constructs onto GPUs.
- Better support for speculative parallelism.
- Support for controlled use of mutation for shared data structures.
- Ongoing work to improve sequential performance.

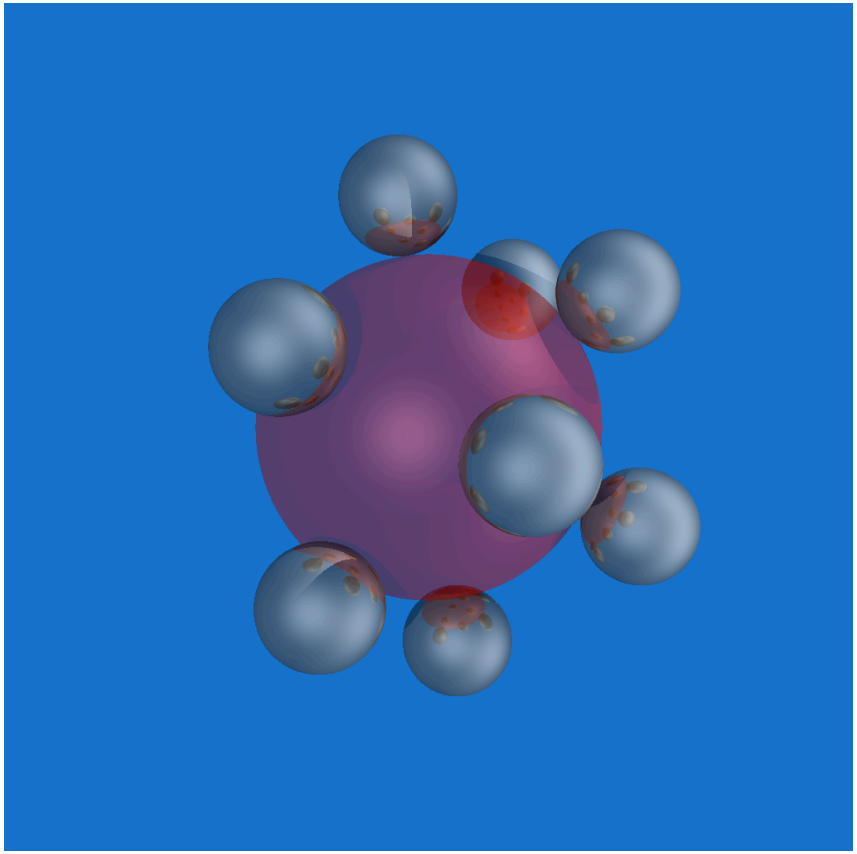
Future work

- Applications: interactive graphics, computer vision, and medical imaging.
- Mapping NDP constructs onto GPUs.
- Better support for speculative parallelism.
- Support for controlled use of mutation for shared data structures.
- Ongoing work to improve sequential performance.

Future work

- Applications: interactive graphics, computer vision, and medical imaging.
- Mapping NDP constructs onto GPUs.
- Better support for speculative parallelism.
- Support for controlled use of mutation for shared data structures.
- Ongoing work to improve sequential performance.

Questions?



http://manticore.cs.uchicago.edu